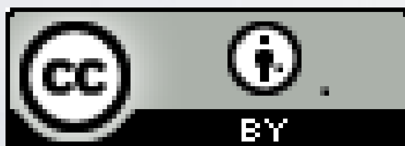
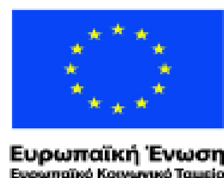
ΕΛΛΗΝΙΚΗ ΔΗΜΟΚΡΑΤΙΑ
ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΡΗΤΗΣ

Εισαγωγή στον Προγραμματισμό
Introduction to Programming

Διάλεξη 22: Δοκιμές

Γ. Παπαγιαννάκης



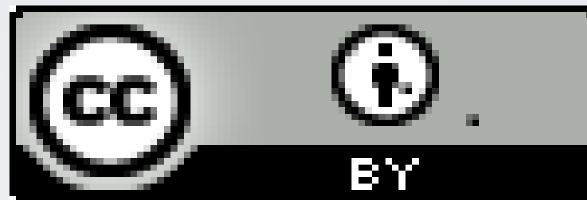


ΥΠΟΥΡΓΕΙΟ ΠΑΙΔΕΙΑΣ & ΘΡΗΣΚΕΥΜΑΤΩΝ, ΠΟΛΙΤΙΣΜΟΥ & ΑΘΛΗΤΙΣΜΟΥ
ΕΙΔΙΚΗ ΥΠΗΡΕΣΙΑ ΔΙΑΧΕΙΡΙΣΗΣ
Με τη συγχρηματοδότηση της Ελλάδας και της Ευρωπαϊκής Ένωσης

Άδειες Χρήσης

- Το παρόν εκπαιδευτικό υλικό υπόκειται στην άδεια χρήσης **Creative Commons** και ειδικότερα

Αναφορά Δημιουργού 3.0 - Μη εισαγόμενο Ελλάδα
(Attribution 3.0– Unported GR)



- Για εκπαιδευτικό υλικό, όπως εικόνες, που υπόκειται σε άλλου τύπου άδειας χρήσης, η άδεια χρήσης αναφέρεται ρητώς.

Χρηματοδότηση

- Το παρόν εκπαιδευτικό υλικό έχει αναπτυχθεί στα πλαίσια του εκπαιδευτικού έργου του διδάσκοντα.
- Το έργο «**Ανοικτά Ακαδημαϊκά Μαθήματα στο Πανεπιστήμιο Κρήτης**» έχει χρηματοδοτήσει μόνο τη αναδιαμόρφωση του εκπαιδευτικού υλικού.
- Το έργο υλοποιείται στο πλαίσιο του Επιχειρησιακού Προγράμματος «Εκπαίδευση και Δια Βίου Μάθηση» και συγχρηματοδοτείται από την Ευρωπαϊκή Ένωση (Ευρωπαϊκό Κοινωνικό Ταμείο) και από εθνικούς πόρους.

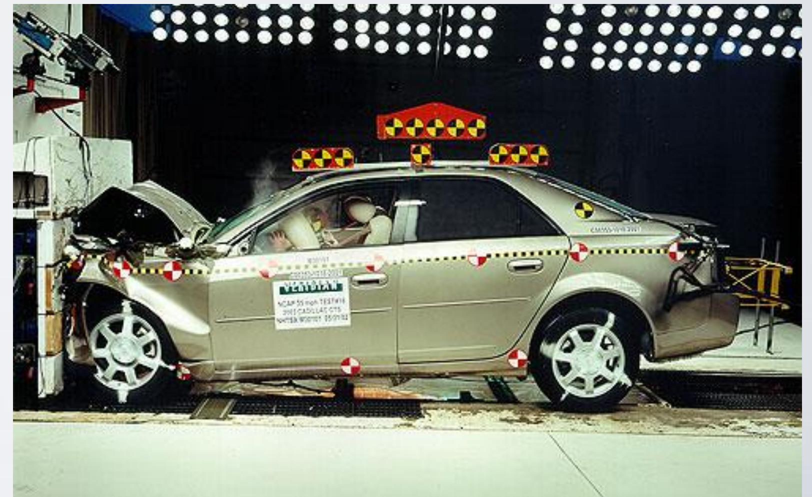
HY-150 Προγραμματισμός
CS-150 Programming

Lecture 22:
Testing

G. Papagiannakis

Abstract

- This lecture is an introduction to the design and testing of program units (such as functions and classes) for correctness. We discuss the use of interfaces and the selection of tests to run against them. We emphasize the importance of designing systems to simplify testing and testing from the start. Proving programs correct and performance problems are also briefly considered.



Overview

- Correctness, proofs, and testing
- Dependencies
- System tests
- Testing GUIs
- Resource management
- Unit and system tests
- Finding assumptions that do not hold
- Design for testing
- Performance

Correctness

- Questions to ask about a program
  - Is your program correct?
  - What makes you think so?
  - How sure are you?
  - Why?
  - Would you fly in a plane that depended on that code?
- You have to be able to reason about your code to have any real certainty
  - Programming is generally unsystematic
  - Debugging is generally unsystematic
  - What are you willing to bet that you found the last bug?
- Related interesting questions
  - Could the program run forever if the hardware didn't fail?
  - Does it always deliver its results in a reasonable time?

Proofs

- So why not just prove mathematically that our program is correct?
  - It's often too hard and/or takes too long
  - Sometimes proofs are wrong too (even proofs produced by computers or by experts!).
  - Computer arithmetic isn't the same as "real" math—remember the rounding and overflow errors we saw (due to finite and limited precision)?
  - So we do what we can: follow good design principles, test, test, and then test some more!

Testing

- "A systematic way to search for errors"
- Real testers use a lot of tools
  - Unit test frameworks
  - Static code analysis tools
  - Fault injection tools
  - …
- When done well, testing is a highly skilled and most valuable activity
- "Test early and often"
  - Whenever you write a function or a class, think of how you might test it
  - Whenever you make a significant change, re-test
  - Before you ship (even after the most minor change), re-test

Testing

- Some useful sets of values to check (especially boundary cases):
  - the empty set
  - small sets
  - large sets
  - sets with extreme distributions
  - sets where “what is of interest” happens near the ends
  - sets with duplicate elements
  - sets with even and with odd number of elements
  - some sets generated using random numbers

Primitive test harness for binary_search()

```
int a1[ ] = { 1,2,3,5,8,13,21 };
if (binary_search(a1,a1+sizeof(a1)/sizeof(*a1),1) == false)
   cout << "1 failed";
if (binary_search(a1,a1+sizeof(a1)/sizeof(*a1),5) == false)
   cout << "2 failed";
if (binary_search(a1,a1+sizeof(a1)/sizeof(*a1),8) == false)
   cout << "3 failed";
if (binary_search(a1,a1+sizeof(a1)/sizeof(*a1),21) == false)
   cout << "4 failed";
if (binary_search(a1,a1+sizeof(a1)/sizeof(*a1),-7) == true)
   cout << "5 failed";
if (binary_search(a1,a1+sizeof(a1)/sizeof(*a1),4) == true)
   cout << "6 failed";
if (binary_search(a1,a1+sizeof(a1)/sizeof(*a1),22) == true)
   cout << "7 failed";
```

A Better Test (still primitive)

Put the variables into a data file, *e.g.*, with a format of

{ 27 7 { 1 2 3 5 8 13 21} 0 }

meaning

{test_number value {sequence} result}

i.e., test #**27** calls our binary_search to look for the value **7** in the sequence **{ 1 2 3 5 8 13 21}** and checks that the result is **0** (false, that is, not found).

Now it's (relatively) easy to write lots of test cases, or even write another program to generate a data file with lots of (random) cases.

Dependencies

Basically we want every function to:

- have well-defined inputs
- have well-defined results
  - including any modifications to input parameters
  - in a determinate amount of time (no infinite loops, please)
- not have dependencies on objects that are not its explicit inputs
  - Hard to achieve in real life
- not use more resources than are available and appropriate
  - E.g., time, memory, internet bandwidth, files, and locks

Dependencies

How many dependencies can you spot in this nonsense function?

```
int do_dependent(int a, int& b) // messy function
                                // undisciplined dependencies
{
   int val ;
   cin>>val;
   vec[val] += 10;
   cout << a;
   b++;
   return b;
}
```

Resource Management

What resources (memory, files, etc.) acquired may not always be properly released in this nonsense function?

```
void do_resources1(int a, int b, const char* s)   // messy function
                                                  // undisciplined resource use
{
    FILE* f = fopen(s,"r");          // open file (C style)
    int* p = new int[a];             // allocate some memory
    if (b<=0) throw Bad_arg();       // maybe throw an exception
    int* q = new int[b];             // allocate some more memory
    delete[ ] p;                     // deallocate the memory pointed to by p
}
```

Better Resource Management

```
// less messy function
void do_resources2(int a, int b, const string& s)
{
   istream is(s.c_str(),"r");        // open file
   vector<int>v1(a);                 // create vector (owning memory)
   if (b<=0) throw Bad_arg();        // maybe throw an exception
   vector<int> v2(b);                // create another vector (owning memory)
}
```

Can do_resources2() leak anything?

Loops

Most errors occur at the ends, *i.e.*, at the first case or the last case. Can you spot 3 problems in this code? 4? 5?

```
int do_loop(vector<int>& vec) // messy function
                              // undisciplined loop
{
   int i;
   int sum;
   while(i<=vec.size()) sum+=v[i];
   return sum;
}
```

Buffer Overflow

- Really a special type of loop error, *e.g.*, “storing more bytes than will fit” into an array—where *do* the “extra bytes” go? (probably not a good place)
- The premiere tool of virus writers and “crackers” (evil hackers)
- Some vulnerable functions (best avoided):
  - **gets, scanf** *// these are the worst: avoid!*
  - **sprintf**
  - **strcat**
  - **strcpy**
  - …

Buffer overflow

- Don't avoid unsafe functions just as a fetish
  - Understand what can go wrong and don't just write equivalent code
  - Even unsafe functions (e.g. **strcpy()**) have uses
    - if you really want to copy a zero terminated string, you can't do better than **strcpy()** – just be sure about your "strings" (How?)

```
char buf[MAX];
char* read_line()          // harmless? Mostly harmless? Avoid like the plague?
{
    int i = 0;
    char ch;
    while (cin.get(ch) && ch!='\n')  buf(i++)=ch;
    buf[i+1]=0;
    return buf;
}
```

Buffer overflow

- Don't avoid unsafe functions just as a fetish
  - Understand what can go wrong and don't just write equivalent code
  - Write simple and safe code

```
string buf;
getline(cin,buf);       // buf expands to hold the newline terminated input
```

Branching

- In **if** and **switch** statements
  - Are all alternatives covered?
  - Are the right actions associated with the right conditions?
- Be careful with nested **if** and **switch** statements
  - The compiler ignores your indentation
  - Each time you nest you must deal with all possible alternatives
    - Each level multiplies the number of alternatives (not just add)
- For **switch** statements
  - remember the **default** case and to **break** after each other case
    - unless you really meant to “fall through”

Branching test (if)

```
void do_branch1(int x, int y)    // messy function
        // undisciplined use of if
{
        if (x<0) {
                if (y<0)
                        cout << "very negative\n";
                else
                        cout << "somewhat negative\n";
        }
        else if (x>0) {
                if (y<0)
                        cout << "very positive\n";
                else
                        cout << "somewhat positive\n";
        }
}
```

Branching test (switch)

```
void do_branch1(int x, int y)    // messy function
        // undisciplined use of switch
{
        if (y<0 && y<=3)
                switch (x) {
                case 1:
                        cout << "one\n";
                        break;
                case 2:
                        cout << "two\n";
                case 3:
                        cout << "three\n";
                }
}
```

System Tests

- Do unit tests first, then combinations of units, and so on, till we get to the whole system
  - Ideally, in isolation from other parts of the system
  - Ideally, in a repeatable fashion
- What about testing GUI based applications?
  - Control inversion makes GUI testing difficult
  - Human behavior is not exactly repeatable
    - Timing, forgetfulness, boredom, etc.
    - Humans still needed at some point (only a human can evaluate “look and feel”)
  - Simulate user input from a test script
    - That way a test harness script takes the place of the human for many tests
  - An excellent application of “layering” with well-defined interfaces between the layers
  - Allows for portability of applications across GUI systems
    - A GUI is often used as a lock-in mechanism

Testing Classes

- A type of unit test
  - but most class objects have state
  - Classes often depend on interactions among member functions
- A base class must be tested in combination with its derived classes
  - Virtual functions
  - Construction/initialization is the combined responsibility of several classes
  - Private data is really useful here (beware of protected data members)
- Take our **Shape** class as an example:
  - **Shape** has several functions
  - A **Shape** has a mutable state (we can add points, change color, etc.); that is, the effect of one function can affect the behavior of another function
  - **Shape** has **virtual** functions; that is, the behavior of a **Shape** depends on what (if any) class has been derived from it
  - **Shape** is not an algorithm (why not?)
  - A change to a **Shape** can have an effect on the screen (so maybe we still need a human tester?)

Testing a Shape (through a Line)

```
Line ln(Point(10,10), Point(100, 100));
ln.draw();                          // see if it appears

// check the points:
if (ln.number_of_points() != 2) cerr << "wrong number of points";
if (ln.point(0)!=Point(10,10)) cerr<< "wrong point 1";
if (ln.point(1)!=Point(100,100)) cerr<< "wrong point 2";

for (int i=0; i<10; ++i) {          // see if it moves
        ln.move(i+5,i+5);
        ln.draw();
}
```

```
for (int i=0; i<10; ++i) {          // see if it moves back to where it started
        ln.move(i–5,i–5);
        ln.draw();
}
if (point(0)!=Point(10,10)) cerr<< "wrong point 1 after move";
if (point(1)!=Point(100,100)) cerr<< "wrong point 2 after move";

for (int i = 0; i<100; ++i) {       // see if the color changes correctly
        ln.set_color(Color(i*100));
        if (ln.color() != i*100) cerr << "bad set_color";
        ln.draw();
}

for (int i = 0; i<100; ++i) {       // see if the style changes correctly
        ln.set_style(Line_style(i*5));
        if (ln.style() != i*5) cerr << "bad set_style";
        ln.draw();
}
```

Finding assumptions that do not hold

- For example, illegal input arguments
  - Should never happen, but it does
- Check before each call or at the beginning of the function
  - Depending on which code we can modify
  - *E.g.*, **sqrt** first checks that its argument is a non-negative value
- That can be difficult/problematic:
  - Consider **binary_search(a,b,v);** // is v in [a:b)
    - For forward iterators (*e.g.*, for a **list**), we can't test if **a<b** – no < operation
    - For random-access iterators ,we can't check if a and b are part of the same sequence
    - The only perfect solution involves a run-time checking library
  - Scanning the entire sequence to verify it's sorted is much more work than actually doing the binary search
    - The purpose of binary_search() is to be faster than linear search
- Sometimes, check in "debug/test mode" only
  - Leave (only) affordable tests in production code

Design for Testing

- Use well-defined interfaces
  - so that you can write tests for the use of these interfaces
  - Define invariants, pre- and post conditions
- Have a way of representing operations as text
  - so that they can be stored, analyzed and replayed
- Embed tests of unchecked assumptions (assertions) in the calling and/or called code
  - to catch bad arguments before system testing
- Minimize dependencies and keep dependencies explicit
  - To make it easier to reason about the code
- Have a clear resource management strategy

This will also minimize debugging!

Debugging

- Is a technique and attitude
  - Please revisit Lecture 4 notes
- Both debugging and testing catch bugs, but are they the same?
  - No!
  - Debugging is ad hoc
  - Debugging is concerned on removing known bugs and implementing features
- Usually we like testing but we hate debugging!
- Good early unit testing and design for testing minimizes debugging!

Performance

- Is it efficient enough?
  - Note: ***Not*** "Is it as efficient as possible?"
  - Computers are fast: You'll have to do millions of operations to even notice (without using tools)
  - Accessing permanent data (on disc) repeatedly can be noticed
  - Accessing the web repeatedly can be noticed
- Time "interesting" test cases
  - *e.g.*, using **time** or **clock()**
  - Repeat ≥3 times; should be ± 10% to be believable

Performance

- What's wrong with this?

```
for (int i=0; i<strlen(s); ++i)  {
        // do something with s[i]
}
```

- It was part of an internet message log analyzer
  - Used for files with many thousands of long log lines

Timing

- How do we know if a piece of code is fast enough?
- How do you know how long an operation takes?
- What if you want to measure something that takes just milliseconds?
- What if you want to do your own, more detailed, measurements of a part of a program?

Using **clock()**

```
#include <ctime>
#include <iostream>
 using namespace std;
int main(){
int n = 10000000;                  // repeat do_somenting() n times
clock_t t1 = clock();              // start time
if (t1 = = clock_t(-1)) {                    // clock_t(-1) means “clock() didn't work”
    cerr << "sorry, no clock\n";
    exit(1);
}
for (int i = 0; i<n; i++) do_something(); // timing loop
clock_t t2 = clock();                        // end time
if (t2 = = clock_t(-1)) {
    cerr << "sorry, clock overflow\n";
    exit(2);
}
cout << "do_something() " << n << " times took "
    << double(t2-t1)/CLOCKS_PER_SEC << " seconds "      // scale result
    << " (measurement granularity: "
    << CLOCKS_PER_SEC << " of a second)\n"; }
```

Acknowledgements

Bjarne Stroustrup

Programming -- Principles and Practice Using C++

http://www.stroustrup.com/Programming/

Thank you!